The Self-Testing COTS Components (STECC) Strategy – a new form of improving component testability*

Sami Beydeda and Volker Gruhn
University of Leipzig
Department of Computer Science
Chair of Applied Telematics / e-Business
Klostergasse 3
04109 Leipzig, Germany
{sami.beydeda,volker.gruhn}@informatik.uni-leipzig.de

ABSTRACT
Development of a software system from existing components can surely have various benefits, but can also entail a series of problems. One type of problems is caused by a limited exchange of information between the developer and user of a component. A limited exchange and thereby a lack of information can have various consequences, among them the requirement to test a component prior to its integration into a software system. A lack of information cannot only make test prior to integration necessary, it can also complicate this tasks. However, difficulties in testing can be avoided if certain provisions to increase testability are taken beforehand. This article describes a new form of improving testability of, particularly commercial, components, the *self-testing COTS components (STECC) strategy*.

KEY WORDS
COTS components, self-testing components.

1 Introduction

Quality assurance, including testing, conducted in development and use of a component can be considered according to [8, 7] from two distinct perspectives. These perspectives are those of the *component provider* and *component user*. The component provider corresponds to the role of the developer of a component and the component user to that of a client of the component provider, thus to that of the developer of a system using the component.

The use of components in the development of software systems can surely have several benefits, but can also introduce new problems. Such problems concern, for instance, testing of components. The component provider and component user need to exchange various types of information during the development of the component itself and also during the development of a system using the component. However, exchange of such information can be limited due to various reasons and both the component provider and component user can face a lack of information. Such a lack of information might cause various problems which in turn might require that tests have also to be carried out by the component user. This contradicts to the believe that a component thoroughly tested by the component provider does not need to be retested by the component user. Such a lack of information might not only obligate the component user to test a component, it might also complicate component user's tests. An important example for this is a lack of source code for test case generation purposes. Limited exchange of information among the component provider and component user is to our opinion the main reason why testing of components is a research problem of its own and needs to be considered in particular.

*The chair of Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

2 Problems due to a limited exchange of information

The component provider and component user generally need to exchange information during the various phases of developing the component and a component-based system using the component. As shown in [1, 2], this exchange of information can suffer due to various reasons and both the component provider and the component user can face certain problems.

2.1 Context-dependent development of a component

One type of information required for the development of a component is that indicating the application environment in which it will later be used. Such information, however, might not be available so that the component provider might develop the component on the basis of assumptions concerning the application environment. The component is then explicitly designed and developed for the needs of the assumed application environment, which, however, might not be the one in which the it will be actually used. Even if the component is not tailored to a certain application environment but constructed for the broader market, the

component provider might unconsciously assume a certain application environment and its development might again become context-dependent. A consequence of context-dependent development of a component can be that testing is also conducted context-dependently. A component might work well in a certain application environment and can exhibit failures in another [17, 15].

The results of a case study given in [17] show the problem of context-dependent testing in practice. A component has been considered as part of two different component-based systems. The component-based systems have mainly provided the same functionality, but differed in the operations profiles associated with them. The operational profile of a system is a probability distribution which assigns a probability to each element in the input domain giving the likelihood that this element is entered as input during operation, e.g. [5]. A set of test cases has been generated for the first component-based system as a 98% confidence interval. The probability that an arbitrary input has been tested has thus been 98%, and the component-based system and the component in its context have been considered as being adequately tested. The fact that a component is adequately tested in the context of a system does generally not imply that it is also adequately tested in the context of another. In the case study, the set of test cases has only corresponded to a 24% confidence interval of the second system's operation profile and occurrence of an untested input during the second system's operation has thus been much more likely.

Observations such as those made during the case study are captured by a formal model of test case adequacy given in [12], which is an extension of that in [16] to components and component-based systems. The formal model of test suite adequacy also considers that a component tested with a certain test suite might be sufficiently, i.e. adequately, tested in the context of a certain application environment, but might not be adequately covered by the same test suite in case of another application environment. In the terminology introduced, the test suite is *C-adequate-on-*P_1 with C being the test criterion used to measure test suite adequacy and P_1 the former application environment, but not *C-adequate-on-*P_2 with P_2 being the latter application environment. An exact definition of the formal framework and theoretical investigations on test suite adequacy in case of components and component-based software can be found in [12].

One of the reasons for context-dependent development of a component is often the component provider's lack of information concerning the possible application environments in which the component might be used later. Tests conducted by the component provider might also be context-dependent and a change of application environment, which might be due to reuse of the component, generally requires additional tests in order to give sufficient confidence that the component will behave as intended also in the new application environment. Additional tests are required even if often contrary claimed that components frequently reused need less testing, e.g. [14]. Moreover, a component reused in a new application environment needs to be tested irrespective of its source. A component produced in-house does not necessarily need less testing for reuse purposes than a component being an independent commercial item [17].

2.2 Insufficient documentation of a component

Development of a component-based system generally requires detailed documentations of the components which are to be assembled. Such documentations are usually delivered together with the respective components and each of them needs to include three types of information related to the corresponding component:

> *Functionality.* The specification of the component functionality gives a description of the functions of that component, i.e. its objectives and characteristics actions, to support an user in solving a problem or achieving an objective.
>
> *Quality.* The specification of component quality can address, for instance, of quality assurance, particularly including testing techniques, applied, metrics used to measure quality characteristics and their values.
>
> *Technical requirements.* The specification of the technical requirements of a component needs to address issues such as the resources required, the architectural style assumed, the middleware used.

Documentation delivered together with a component and supposed to include specifications of the above outlined aspects might, however, be insufficient for development of a component-based system. The various types of information provided by the documentation can deviate from those expected syntactically as well as semantically, and it can even be incomplete. This problem can be viewed from two different perspectives. On the one hand, it can be considered as a problem due to a lack of information. The component provider might be suffering from a lack of information and might therefore not provide the information as documentation actually needed by the component user. On the other hand, it can be considered as a reification of a lack of information. Instead assuming the component provider as suffering from a lack of information while developing the component and assembling its documentation, the component user is assumed as suffering from such a lack while developing a component-based system using the component. Insufficient documentation is according to the latter perspective not the effect of a lack of information but its reification. However, the subtle differences of these perspective are not further explored.

A case study which can be found in [6] reports several problems encountered during the integration of four components to a component-based system. The problems encountered are assumed to be caused by assumptions made

during the development of the single components. The assumptions, even concerning the same technical aspects of the components and the component-based system, have been incompatible to each other, which has been the reason for an architectural mismatch as generally referred in the case study to the problems encountered. As one of the solutions to tackle the architectural mismatch problem, the authors propose to make architectural assumptions explicit. These assumptions need to be documented using the appropriate terminology and structuring techniques. Another solution to problems such as those in the case study is proposed in [13]. Here, the author suggests to prototype during component-based development in order to detect such potential problems as one objective of prototyping.

In [9, 10], the authors propose a process model for COTS-based development which includes a specific activity to tackle problems due to insufficient documentation. The process model encompasses an activity called COTS components familiarization in which components selected before are actually used to gain a better understanding of the functionalities available, their quality, and architectural assumptions. Importance of such an activity depends on the quality of the component documentation and decreases with an increase of that.

Both prototyping and familiarization require that the component under consideration is executed, which is also the main characteristic of testing. In fact, both can be considered as testing, if the term of testing is defined more generally without assuming that testing is a quality assurance action. The objectives of both are not necessarily related to quality assurance, but are in principle to obtain information which is not delivered as part of the documentation. Furthermore, components delivered with insufficient documentation might also required testing in its original sense, particularly if the documentation does not include information concerning quality assurance conducted. Even if the documentation includes such information, quality assurance conducted might not be sufficient for the application environment in which the component will be used. In such cases, the component usually needs to be retested also by the component user, since the component user is from the viewpoint of the end-user responsible for the quality of the component-based system and the component user's reputation depends on its quality [17].

2.3 Component user's dependence on the component provider

Context-dependent development and insufficient documentation of a component are two problems resulting by a lack of information which often obligate the component user to test a component before its use. The component user can encounter after the tests are finished and a failure is revealed another problem also due to a lack of information. The problem which the component user can encounter is that of dependence on the component provider. The fault causing the failure often cannot be removed by the component user, since the component user might not have the software artifacts required for isolating and removing the fault. Such artifacts include documentation, test plans and source code of the component, which is usually the case for COTS components. Even if the artifacts required are available to the component user, debugging might be significantly difficult or even impossible due to missing expertise. Missing expertise and insight of the component user might entail significant debugging costs which can even offset the benefits gained by using the component. The component user thereby has often to rely on the component provider for maintenance and support, which the component user, however, might not be able to influence, which gives an uncertainty for the future.

The problem of dependence on the component provider can even aggravate if the component is not maintained as demanded by the component user, or if even the component provider decides to cease support and maintenance or goes bankrupt [17, 15]. The possible financial effects of such an event is shown in [15] on a simple example. It has been suggested to create escrow agreements and protective licensing options for the relevant artifacts of a component to avoid the problems in the above case. Even if the component provider accepts such an agreement, the problems due to missing expertise can still hinder the component user from carrying out the corresponding tasks.

Difficulties for the component user by a dependence on the component provider are not necessarily restricted to maintenance and support. Generally, several of the decisions taken by the component provider during the lifecycle of the component also impact its use as part in a component-based system. Other problems which can occur due to the dependence on the component provider can also be [9, 10, 15]:

- Difficulties to meet deadlines because of delays in releasing a component version,
- some functionality might be promised but never be implemented in the component,
- modifications might have adverse effects, such as incompatibilities or even faults,
- some functionality of the component might not remain as exactly required,
- documentation might be incomplete or might not sufficiently cover modifications,
- technical support offered by the component provider might not be sufficient.

Similar to the problem of context-dependent component development, the problem of component user dependence to the provider varies with the degree of information availability. The more information concerning the component used is available to its user, the less dependent is the

user from the provider, as some of maintenance and support tasks can be carried out by the user. Specifically, the dependence on the provider also affects reputation of the user towards clients. In case of a problem, user's reputation will suffer particularly if the user cannot provide the necessary solutions, even if the problem is caused by the component the provider is responsible for [17].

3 STECC strategy in component testing

3.1 Approach to tackle a lack of information

A lack of information might require the testing of a component by its user prior to its integration in a system, and might significantly complicate this task at the same time. The component user might not possess the information required for this task. Theoretically, the component user can test a component by making certain assumptions and approximating the information required. Such assumptions, however, are often too imprecise to be useful. For instance, control-dependence information can be approximated in safe-critical application contexts by conservatively assuming that every component raises an exception, which is obviously too imprecise and entails a higher testing effort than necessary [8, 7].

Even though often claimed, source code as one type of information often required for testing purposes is not required by itself for testing purposes. It often acts as the source for obtaining other information, such as that concerning control-dependence. Instead making source code available to allow the generation of such information, the information required can also be directly delivered to the component user, obviating source code access. This type of information is often referred to as *meta-information* [11]. Even though the information required might already be available from own testing activities, the component provider might nevertheless not deliver this information to the component user. One reason may be that detailed information, including parts of the source code, can be deduced from it depending on the granularity of the meta-information. Therefore, there is a natural boundary limiting the level of detail of the information deliverable to the user. For some application contexts, however, the level of detail might be insufficient and the component user might not be able to test the component according to certain quality requirements.

The underlying strategy of the method proposed differs from those discussed thus far. Instead of providing the component user with information required for testing, component user tests are supported by the component explicitly. The underlying strategy of the method is to augment a component with functionality specific to testing tools. A component possessing such functionality is capable of testing its own methods by conducting some or all activities of the component user's testing processes, it is thus *self-testing*. The method is thereby called the *self-testing COTS components* (STECC) method [1, 2]. Self-testability does not obviate the generation of detailed technical information. In fact, this information is generated by the component itself during runtime and is internally used in an encapsulated manner. The information generated is transparent to the component user and can thus be more detailed than in the case above. Consequently, tests carried out by the component user through the self-testing capability can thereby be more thorough as in the case of meta-information. Self-testability allows the component user to conduct tests and does not require the component provider to disclose source code or other detailed technical information. It thereby meets the demands of both parties. The overall objective of the STECC method is to provide the necessary means to both parties, the component provider and component user, to enhance a component with this capability and to use it for program-based testing, respectively.

3.2 Impact on the component user's testing processes

The STECC strategy, if considered for testing purposes by the component user, impacts several activities of a component user's testing process. In particular, the single activities of a typical testing process are impacted as follows:

Test plan definition. Some of the decisions made during definition of test plans are addressed by conventions of the STECC strategy and its actual implementation. Such decisions concern, for instance, the target component model and framework. The actual implementation might assume, for instance, the Enterprise JavaBeans component model and framework [4]. Another decision can concerns the technique used for analysis and testing purposes, such as the test case generation technique. Related to test case generation, the actual implementation of the STECC strategy might also prescribe a certain type of completion criterion used to measure testing progress.

Test case generation. Generation of test cases is the integral constituent of self-testability as assumed by the STECC strategy. Test case generation needs to be entirely conducted by the self-testing component due to a lack of its source code and necessary white-box information to the component user, who therefore cannot carry out this task. Various types of test case generation techniques can be embedded in the actual implementation of the STECC strategy. Test cases as generated in this context do not necessarily include expected results. This depends on the fact whether the specification of the component is available in a form in which it can automatically processed.

Test driver and stub generation. The component user does not need to generate test drivers for component method testing. The actual implementation of the STECC strategy usually includes the necessary provisions to execute the methods of the component con-

sidered. Stubs, however, might be necessary if the method to be tested needs to invoke those of absent components. A component can often be embedded in a wide variety of application contexts and the specific application context can therefore often not be anticipated. The component user needs either to provide the stubs or to embed the component in the target application context.

Test execution. The execution of the methods under consideration with generated test cases can also be conducted by the implementation of the STECC strategy. As one possibility for test execution, a dynamic technique can be used for test case generation purposes, which iteratively approaches to appropriate test cases and successively executes the method to be tested for this purpose. As another possibility for test execution, test cases generated can be stored and executed in a separate testing phase.

Test evaluation. The evaluation of tests needs either to be addressed by the component user or by the implementation of the STECC strategy. The reason for this is mainly the fact whether the specification or expected results are available to the implementation. In the case in which the specification or expected results are available, this task can be conducted by the implementation of the STECC strategy. Otherwise, expected results have to be determined and compared with those observed during and after test execution by the tester, i.e. component user.

3.3 Comparison to built-in testing approaches

The STECC approach can be compared to built-in testing approaches in the literature. An overview of built-in testing approaches can be found in [1, 3]. Similar as the STECC approach, built-in testing approaches aim at tackling difficulties in testing components caused by a lack of information, difficulties in test case generation in particular. The STECC approach has the same objective and the approaches can thus be directly compared to it. A comparison of them highlights several differences. These differences include the following:

Firstly, the built-in testing approaches are static in that the component user cannot influence the test cases employed in testing. A component which is built-in testing enabled according to one of the approaches explained either contains a predetermined set of test cases or the generation, even if conducted on-demand during runtime, solely depends on parameters which the component user cannot influence. Specifically, the component user cannot specify the adequacy criterion to be used for test case generation. However, the component user might wish to test all components to be assembled with respect to an unique adequacy criterion. Built-in testing approaches, at least those described, do not allow this. The STECC approach does not have such a restriction. Adequacy criteria, even though constrained to control flow criteria, can be freely specified. Note that this restriction of built-in testing approaches depends on the size of the test case set used. A large set of test case generally satisfies more adequacy criteria than a small set, but increase at the same time resource requirements.

Secondly, built-in testing approaches using a predefined test case set generally require more storage than the STECC approach. Specifically, large components with high inherent complexity might require a large set of test cases for their testing. A large set of test cases obviously requires a substantial amount of storage which, however, can be difficult to provide taking into account the storage required in addition for execution of large components. This is also the case if test cases are stored separately from the component. In contrast, the STECC strategy does not require predetermined test cases and does also not store the generated test case.

Thirdly, built-in testing approaches using a predefined test case set generally require less computation time at component user site. In such a case, the computations for test case generation were already conducted by the component provider and obviously do not have to be repeated by the component user, who thus can save resources, particularly computation time, during testing. Savings in computation time are even magnified if the component user needs to frequently conduct tests, for instance, due to volatility of the technical environment of the component. Storage and computation time consumption of a built-in testing enable component obviously depends on the implementation of the corresponding capabilities and the component provider needs to decide between the two forms of implementation, predefined test case set or generation on-demand, carefully in order to ensure a reasonable trade-off.

The STECC approach has several benefits which makes it more preferable than one of the built-in testing approaches. Except computation time consumption, the STECC approach dominates the three explained built-in testing approaches and it should be the first choice if a component is to be enhanced to facilitate component user's test. However, in some circumstances, computation time might be crucial so that built-in testing approaches, particularly those which do not generate test case on-demand but contain a fixed set of test case, are more preferable. Such a circumstance might be, as mentioned, one in which the technical environment of the component is volatile.

4 Conclusions and future research

The STECC strategy addresses the needs of both the component provider and component user. A situation particularly encountered in the case of commercial components, thus COTS components, is that the component provider might not wish to disclose information, particularly source code, which the component user might require for testing purposes. Our research started with the observation that existing approaches do not appropriately tackle such a situation.

The STECC strategy is a response to such situations. It allows the component user to test the component and to ensure suitability of the component to the target application context regarding its quality without requiring the component provider to publish specific information. It thus meets the demands of both parties. The STECC strategy can lead to a win-win situation insofar that both the component provider and component user can benefit from it. The benefit of the component user is obvious. The STECC strategy, or more clearly self-testability of a component, can be a valuable factor in competition. This potential benefit of the component provider from the STECC strategy becomes more obvious taking into account the specific type of components which are the most appropriate candidates for STECC self-testability, COTS components.

We would like to invite the reader to participate in an open discussion started to gain a consensus concerning the problems and open issues in testing components. The contributions received so far can be found at `http://www.stecc.de` and new contributions can be made by email to `sami.beydeda@informatik.uni-leipzig.de`.

References

[1] S. Beydeda. *The Self-Testing COTS Components (STECC) Method*. PhD thesis, Universität Leipzig, Fakultät für Mathematik und Informatik, 2003.

[2] S. Beydeda and V. Gruhn. Merging components and testing tools: The self-testing COTS components (STECC) strategy (to appear). In *EUROMICRO Conference Component-based Software Engineering Track*. IEEE Computer Society Press, 2003.

[3] S. Beydeda and V. Gruhn. State of the art in testing components (to appear). In *International Conference on Quality Software (QSIC)*. IEEE Computer Society Press, 2003.

[4] L. G. DeMichiel. Enterprise javabeans specification, version 2.1. Technical report, Sun Microsystems, 2002.

[5] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, 1998.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *International Conference on Software Engineering (ICSE)*, pages 179–185. ACM Press, 1995.

[7] M. J. Harrold. Testing: A roadmap. In *The Future of Software Engineering (special volume of the proceedings of the International Conference on Software Engineering (ICSE))*, pages 63–72. ACM Press, 2000.

[8] M. J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *International ICSE Workshop Testing Distributed Component-Based Systems*, 1999.

[9] M. Morisio, C. Seaman, A. Parra, V. Basili, S. Kraft, and S. Condon. Investigating and improving a COTS-based software development process. In *International Conference on Software Engineering (ICSE)*, pages 32–41. ACM Press, 2000.

[10] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon. COTS-based software development: Processes and open issues. *The Journal of Systems and Software*, 61(3):189–199, 2002.

[11] A. Orso, M. J. Harrold, and D. Rosenblum. Component metadata for software engineering tasks. In *International Workshop on Engineering Distributed Objects (EDO)*, volume 1999 of *LNCS*, pages 129–144. Springer Verlag, 2000.

[12] D. S. Rosenblum. Adequate testing of component-based software. Technical Report 97-34, University of California, Department of Information and Computer Science, 1997.

[13] M. Sparling. Lessons learned through six years of component-based development. *Communications of the ACM*, 43(10):47–53, 2000.

[14] C. Szyperski. *Component Software Beyond Object Oriented Programming*. Addison-Wesley, 1998.

[15] J. Voas. COTS software: The economical choice? *IEEE Software*, 15(2):16–19, 1998.

[16] E. J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, 1986.

[17] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.